Inode 구조체를 이용한 **File System** 구현

제출일	2024.06.12	이름	김석주(20180552)
과목	운영체제	이름	김지현(20246029)
		이름	김진한(20201578)
		이름	지주원(20236111)

[목차]

1. 서론

A. 프로젝트 개요

이번 프로젝트의 원래 목표는 프로세스들을 생산해 메모리에 있는 파일시스템 명령어를 프로세스들이 접근하여 수행할 수 있도록 하는 것이었는데 실패하여 inode를 이용한 파일 시스템을 설계하고 구현하는 것으로 바꾸었다. inode(index node)는 파일 시스템에서 파일의 메타데이터를 관리하는 중요한 데이터 구조로, 파일 시스템의 효율성과 안정성을 크게 향상시킨다. 이 프로젝트에서는 inode의 기본적인 동작 원리를 이해하고, 이를 바탕으로 파일 시스템을 구현하여 파일의 저장, 검색 및 관리를 효율적으로 처리하는 것을 목표로 하였다. inode를 통해 파일의 권한, 크기 및 데이터 블록 위치 등의 정보를 체계적으로 관리할 수 있으므로 파일 시스템의 성능과 신뢰성이 증가하게 된다. 다음과 같은 주요 단계를 구현할 것이다.

1. **inode** 구조 설계: 파일의 메타데이터를 효율적으로 저장하고 관리할 수 있는 inode 구조를 설계한다.
2. 파일 생성 및 삭제: 파일 생성 시 inode를 할당하고, 파일 삭제 시 inode를 해제하는 메커니즘을 구현한다.
3. 파일 읽기 및 쓰기: 파일의 데이터 블록을 inode를 통해 추적하고, 파일의 내용을 읽고 쓸 수 있는 기능을 구현한다.

4. 디렉토리 관리: 파일 시스템 내의 디렉토리를 관리하고, 디렉토리 내 파일의 inode를 연결하는 구조를 구현한다.

이를 통해 inode 기반 파일 시스템의 기본 개념과 구현 방법을 배우고, 실제 파일 시스템의 효율성과 안정성을 높이는 기술을 습득할 수 있을 것이다.

2. 설계 개요

A. 아키텍쳐 개요

이번 프로젝트에서 구현한 inode 기반 파일 시스템은 파일 메타데이터 관리, 파일 생성 및 삭제, 파일 읽기 및 쓰기 등의 기능을 수행하는 단순화된 파일 시스템이다. 각 기능은 독립적인 모듈로 구현되어 있으며, 이를 통해 파일 시스템의 효율성과 안정성을 향상시킬 수 있도록 설계하였다. 전체 시스템 아키텍처는 다음과 같이 구성되어 있다.

시스템 구성 요소

메타데이터 관리

- **Inode Table** : 파일의 메타데이터를 관리하는 inode들의 집합. 각 inode는 파일 크기, 생성 및 수정 시간 등의 정보를 포함한다.
- **Superblock** : 파일 시스템의 전반적인 정보를 관리하는 구조체. 사용 가능한 inode의 수, 데이터 블록의 수, 파일 시스템의 크기 등을 관리한다.

파일 생성 및 삭제

- **Inode Allocator** : 새로운 파일이 생성될 때 사용 가능한 inode를 할당하고, 파일이 삭제될 때 해당 inode를 해제하는 모듈.
- **Directory Management**: 파일과 디렉토리 간의 관계를 관리. 파일이 생성되거나 삭제될 때 해당 디렉토리의 내용을 업데이트한다.

파일 읽기 및 쓰기

- **File Read/Write Module**: Inode를 통해 파일 데이터 블록에 접근하여 파일의 내용을 읽거나 쓰는 기능을 수행한다. 데이터 블록의 위치 정보는 해당 파일의 inode에 저장된다.

이러한 요소를 통해, inode 기반 파일 시스템의 구조와 작동 원리를 이해하고, 이를 바탕으로 효율적인 파일 관리 시스템을 구현하였다. 파일 시스템의 구성 요소는 miniOS 내에서 작동되도록 설계하였다.

3. 모듈 상세 설명

A. miniOS

miniOS는 기본적인 부트로더, 커널, 디바이스 드라이버, 라이브러리, 그리고 inode 기반 파일 시스템을 포함하고 있다.

miniOS 구조:

```
├── README.md # 프로젝트 설명 및 사용 방법 문서
├── Makefile # 전체 프로젝트 빌드 자동화를 위한 메이크파일
├── boot/ # 부트로더 소스 코드
O └── boot.asm # 부트로더 어셈블리 코드
```

```
├── kernel/ # 커널 소스 코드
O ├── kernel.c # 커널 메인 C 소스 파일
O └── ...
├── drivers/ # 디바이스 드라이버 코드
O ├── keyboard.c # 키보드 드라이버
O ├── screen.c # 화면(비디오) 드라이버
O └── ...
├── lib/ # 커널 라이브러리 및 공통 유틸리티
O ├── stdio.c # 기본 입출력 함수
O ├── string.c # 문자열 처리 함수
O └── ...
├── include/ # 헤더 파일
O ├── kernel.h # 커널 관련 공통 헤더
O ├── drivers/ # 드라이버 헤더 파일
O └── lib/ # 라이브러리 헤더 파일
└── scripts/ # 빌드 및 유틸리티 스크립트
O ├── build.sh # 빌드 스크립트
O └── run_qemu.sh # QEMU를 통해 OS 이미지 실행 스크립트
```

miniOS는 다음과 같은 주요 디렉토리와 파일로 구성된다.

- `README.md`: 프로젝트에 대한 설명과 사용 방법을 담은 문서.
- `Makefile`: 프로젝트의 전체 빌드 과정을 자동화하기 위한 메이크파일.
- `boot/`: 부트로더 소스 코드를 포함하고 있으며 `boot.asm` 파일에는 부트로더의 어셈블리 코드가 들어있다.
- `kernel/`: 커널 소스 코드가 들어있으며 `kernel.c` 파일은 커널의 메인 C 소스 파일이다.
- `drivers/`: 디바이스 드라이버 코드를 포함하고 있다. 예를 들어 `keyboard.c`, `screen.c` 등이 있다.

- `lib/`: 커널 라이브러리 및 공통 유틸리티 함수를 포함하고 있으며 `stdio.c`, `string.c` 등의 파일이 있다.
- `include/`: 프로젝트의 헤더 파일들을 포함하고 있으며 커널, 드라이버, 라이브러리 각각의 헤더 파일들이 있다.
- `scripts/`: 빌드 및 유틸리티 스크립트를 포함하고 있으며, `build.sh`, `run_qemu.sh` 등이 있다.

기능 설명

- 부트로더: 시스템이 부팅될 때 가장 먼저 실행되는 코드로, 커널을 메모리로 로드하고 실행한다.
- 커널: 시스템의 핵심을 이루며, 프로세스 관리, 메모리 관리, 파일 시스템 관리 등의 주요 기능을 담당한다.
- 디바이스 드라이버: 키보드, 화면 등과 같은 하드웨어 장치와의 통신을 담당한다.
- 라이브러리: 표준 입출력, 문자열 처리 등의 기본적인 기능을 제공하는 라이브러리 함수들을 포함한다.
- **inode** 기반 파일 시스템: 파일과 디렉토리의 메타데이터를 관리하고, 파일 생성, 삭제, 읽기, 쓰기 등의 기능을 제공한다.

B. inode 기반 파일 시스템

위 miniOS에 새롭게 추가될 inode 기반 파일 시스템 모듈은 kernel 폴더에 추가되며 파일과 디렉토리의 메타데이터를 관리하는 방식이다. 각 파일 또는 디렉토리는 고유한 inode 번호를 가지며, inode는 파일의 권한, 크기, 데이터 블록 위치 등의 정보를 포함한다.
inode 기반 파일 시스템은 트리 구조를 사용하여 디렉토리와 파일을 관리하는 방식이다.

시스템을 구현하기 위해 7개의 구조체를 정의했다.

1. Inode 구조체

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <time.h> // 파일 시간 정보를 위해 추가
#define MAX_INODES 100

typedef struct Inode {
    int fileSize; // 파일 크기
    time_t created; // 파일 생성 시간
    time_t modified; // 파일 수정 시간
    int linkCount; // 링크 수
} Inode;
```

Inode는 파일과 디렉토리의 메타데이터를 저장하는 구조체이다.

- **fileSize**: 파일 크기
- **created**: 파일 또는 디렉토리 생성 시간
- **modified**: 파일 또는 디렉토리가 마지막으로 수정된 시간
- **linkCount**: 파일 또는 디렉토리에 대한 하드 링크의 개수

2. InodeTable 구조체

```c
typedef struct InodeTable {
    Inode inodes[MAX_INODES];
    bool isAllocated[MAX_INODES];
} InodeTable;
InodeTable inodeTable;
```

- **inodes**: Inode 구조체의 배열. 각 파일 또는 디렉토리에 대한 메타데이터를 저장한다.
- **isAllocated**: inodes 배열의 각 인덱스의 할당 여부를 나타내는 boolean 배열

3. Superblock 구조체

```c
typedef struct Superblock {
    int totalInodes;
    int usedInodes;
    int totalBlocks;
    int usedBlocks;
    int fileSystemSize;
} Superblock;
Superblock superblock;
```

Superblock은 파일 시스템 전체에 대한 정보를 저장한다.

- **totalInodes**: 파일 시스템에 존재할 수 있는 최대 inode 개수
- **usedInodes**: 현재 사용 중인 inode의 개수
- **totalBlocks**: 파일 시스템에 존재할 수 있는 최대 블록 개수
- **usedBlocks**: 현재 사용 중인 블록의 개수
- **fileSystemSize**: 파일 시스템의 전체 크기

4. Directory 구조체

```c
typedef struct Directory {
    char name[100]; // 디렉터리 이름
    void* children[30]; // 자식 노드 포인터 배열 (디렉터리 또는 파일), 최대 30개로 제한
    int childCount; // 현재 자식 노드의 수
    int inodeIndex;
    Inode inode; // 디렉터리의 inode 정보
} Directory;
```

Directory는 디렉토리의 이름과 자식 노드(디렉토리 또는 파일)를 저장하는 구조체이다.

- **name**: 디렉토리 이름
- **children**: 자식 노드의 포인터 배열 (최대 30개)
- **childCount**: 현재 자식 노드의 수
- **inodeIndex**: 디렉토리의 inode 인덱스
- **inode**: 디렉토리의 메타데이터를 저장하는 Inode 구조체

5. File 구조체

```c
typedef struct File {
    char name[100]; // 파일 이름
    char content[1024]; // 파일 내용
    int inodeIndex;
    Inode inode; // 파일의 inode 정보
} File;
```

File은 파일의 이름과 내용을 저장하는 구조체이다.

- **name**: 파일 이름
- **content**: 파일 내용
- **inodeIndex**: 파일의 inode 인덱스
- **inode**: 파일의 메타데이터를 저장하는 Inode 구조체

6. NodeType 열거형

```c
typedef enum { DIR_TYPE, FILE_TYPE } NodeType;
```

NodeType은 노드가 디렉토리인지 파일인지 구분하기 위한 열거형이다.

- **DIR_TYPE**: 디렉토리 타입
- **FILE_TYPE**: 파일 타입

7. Node 구조체

```c
typedef struct Node {
    NodeType type; // 노드 타입 (디렉터리 또는 파일)
    int inode; // 새로운 멤버 추가
    union {
        Directory dir;
        File file;
    };
    struct Node* parent; // 부모 노드 포인터
} Node;
```

Node는 파일 시스템의 기본 단위로 디렉토리 또는 파일을 나타낸다.

- **type**: 노드 타입 (DIR_TYPE 또는 FILE_TYPE)
- **inode**: 노드의 inode 인덱스
- **dir** 또는 **file**: 디렉토리 또는 파일에 대한 정보 (공용체 사용)
- **parent**: 부모 노드 포인터

이 구조체를 이용한 함수들을 자세하게 설명하도록 하겠다.

- **allocateInode**

```
int allocateInode() {
    for (int i = 0; i < MAX_INODES; i++) {
        if (!inodeTable.isAllocated[i]) {
            inodeTable.isAllocated[i] = true;
            superblock.usedInodes++;
            printf("Inode %d 가 할당되었습니다.\n", i);
            return i;
        }
    }
    printf("더 이상 할당 가능한 inode가 없습니다.\n");
    return -1;
}
```

파일 시스템에서 사용할 수 있는 inode를 할당한다. MAX_INODES까지 반복하며 사용 가능한 inode를 찾고, 찾았다면 해당 inode를 할당하고 사용 중인 inode 수에 해당하는 superblock 구조체의 변수를 업데이트한다. 할당된 inode의 인덱스 또는 실패 시 -1을 반환한다.

- **createNode**

```
Node* createNode(const char* name, NodeType type, Node* parent) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->type = type;
    newNode->parent = parent;

    int inodeIndex = allocateInode();
    if (inodeIndex == -1) {
        printf("더 이상 할당 가능한 inode가 없습니다.\n");
        free(newNode);
        return NULL;
    }
    time_t currentTime = time(NULL);

    if (type == DIR_TYPE) {
        strcpy(newNode->dir.name, name);
        newNode->dir.childCount = 0;
        newNode->dir.inodeIndex = inodeIndex;
        inodeTable.inodes[inodeIndex].fileSize = 0;
        inodeTable.inodes[inodeIndex].created = currentTime;
        inodeTable.inodes[inodeIndex].modified = currentTime;
        inodeTable.inodes[inodeIndex].linkCount = 0;

        newNode->dir.inode = inodeTable.inodes[inodeIndex];
    } else {
        strcpy(newNode->file.name, name);
        memset(newNode->file.content, 0, sizeof(newNode->file.content));
        newNode->file.inodeIndex = inodeIndex;
        inodeTable.inodes[inodeIndex].fileSize = strlen(newNode->file.content);
        inodeTable.inodes[inodeIndex].created = currentTime;
        inodeTable.inodes[inodeIndex].modified = currentTime;
        inodeTable.inodes[inodeIndex].linkCount = 1;

        newNode->file.inode = inodeTable.inodes[inodeIndex];
    }

    return newNode;
}
```

매개변수:

- **name**: 노드의 이름
- **type**: 노드 타입(DIR_TYPE 또는 FILE_TYPE)
- **parent**: 부모 노드

새로운 노드 (파일 또는 디렉토리)를 생성하고 초기화한다. Node 구조체 메모리를 동적 할당하고 새로운 inode를 할당한다. inode 할당을 실패하면 노드를 해제하고 NULL을 반환한다. 노드 타입에 따라

메타데이터 초기화 작업을 수행한다.
디렉토리의 경우 이름, 자식 수, inode 및 inodetable 정보를 초기화하고 파일의 경우 이름, 내용, inode 및 inodetable 정보를 초기화한다. 초기화된 Node 포인터가 반환된다.

- **freeInode**

```
void freeInode(int index) {
    if (index >= 0 && index < MAX_INODES) {
        inodeTable.isAllocated[index] = false;
        superblock.usedInodes--;
        printf("Inode %d 가 해제되었습니다.\n", index);
    }
}
```

매개변수:
- **index**: 해제할 inode의 인덱스

할당된 inode를 해제한다. index가 유효한 범위인지 확인하고 inode를 사용 가능한 상태로 만든 뒤 사용 중인 inode 수에 해당하는 superblock 구조체 변수를 업데이트한다.

- **addChild**

```
void addChild(Node* parent, Node* child) {
    if (parent->dir.childCount < 30) {
        parent->dir.children[parent->dir.childCount++] = child;
    } else {
        printf("자식 노드의 최대 개수를 초과했습니다.\n");
    }
}
```

매개변수:
- **parent**: 자식 노드를 추가할 부모 노드
- **child**: 추가할 자식 노드

이 함수는 부모 디렉토리의 자식 노드 배열에 자식 노드를 추가한 뒤 자식 수를 증가시킨다. 자식 노드가 30개를 초과할 경우 오류 메시지를

출력한다.

- **printTree**

```
void printTree(Node* node, int level) {
    for (int i = 0; i < level; i++) {
        printf("  ");
    }
    if (node->type == DIR_TYPE) {
        printf("%s/\n", node->dir.name);
    } else {
        printf("%s\n", node->file.name);
    }
    if (node->type == DIR_TYPE) {
        for (int i = 0; i < node->dir.childCount; i++) {
            printTree((Node*)node->dir.children[i], level + 1);
        }
    }
}
```

매개변수:

- **node**: 출력할 트리의 루트 노드
- **level**: 현재 노드의 깊이 (들여쓰기를 위해 사용)

각 노드의 이름을 파일 시스템의 트리 구조로 출력한다. 현재 노드의 이름을 출력하며 디렉토리일 경우 재귀적으로 자식 노드들을 출력한다.

- **freeTree**

```
void freeTree(Node* node) {
    if (node->type == DIR_TYPE) {
        for (int i = 0; i < node->dir.childCount; i++) {
            freeTree((Node*)node->dir.children[i]);
        }
    }
    freeInode(node->type == DIR_TYPE ? node->dir.inodeIndex : node->file.inodeIndex);
    free(node);
}
```

매개변수:

- **node**: 해제할 트리의 루트 노드

트리 구조를 재귀적으로 해제한다. 디렉토리일 경우 먼저 자식 노드들을 재귀적으로 해제하고 최종적으로 현재 노드의 inode와 메모리를 해제한다.

- **findNode**

```
Node* findNode(Node* node, const char* name, NodeType type) {
    if (node->type == type && strcmp(node->dir.name, name) == 0) {
        return node;
    }
    if (node->type == DIR_TYPE) {
        for (int i = 0; i < node->dir.childCount; i++) {
            Node* found = findNode((Node*)node->dir.children[i], name, type);
            if (found != NULL) {
                return found;
            }
        }
    }
    return NULL;
}
```

매개변수:

- **node**: 검색을 시작할 루트 노드
- **name**: 찾으려는 노드 이름
- **type**: 찾으려는 노드의 타입 (디렉토리 또는 파일)

파일 시스템 트리에서 특정 이름과 타입에 해당하는 노드를 찾는 함수이다. 현재 노드가 찾으려는 노드인지 확인하고, 디렉토리일 경우 자식 노드들을 재귀적으로 검색한다.
반환값은 찾은 노드의 포인터 또는 NULL (찾지 못한 경우) 이다.

- **updateFileContent**

```
void updateFileContent(Node* fileNode, const char* newContent) {
    if (fileNode == NULL || fileNode->type != FILE_TYPE) {
        printf("유효하지 않은 파일 노드입니다.\n");
        return;
    }
    int newContentSize = strlen(newContent);
    if (newContentSize >= sizeof(fileNode->file.content)) {
        printf("파일 내용이 너무 깁니다. 최대 길이는 %lu 입니다.\n", sizeof(fileNode->file.content) - 1);
        return;
    }
    strcpy(fileNode->file.content, newContent);
    int inodeIndex = fileNode->file.inodeIndex;
    inodeTable.inodes[inodeIndex].fileSize = newContentSize;
    inodeTable.inodes[inodeIndex].modified = time(NULL);
}
```

매개변수:

- **fileNode**: 업데이트할 파일 노드에 대한 포인터
- **newContent**: 새로운 파일 내용

파일 내용, 크기, 수정 시간 등을 업데이트하는 함수이다. 파일 노드가 유효한지 확인하고 새로운 내용의 길이가 파일 내용 배열의 크기를 초과하지 않는지 확인한다.

- **readfile**

```
void readfile(Node* node, const char* name) {
    bool found = false;

    if (node->type == FILE_TYPE) {
        if (strcmp(node->file.name, name) == 0) {
            printf("파일 이름: %s\n", node->file.name);
            printf("파일 내용: %s\n", node->file.content);
            printf("파일 크기: %d bytes\n", node->file.inode.fileSize);

            char* createdTime = ctime(&node->file.inode.created);
            createdTime[strlen(createdTime) - 1] = '\0'; // 줄바꿈 제거
            printf("생성 시간: %s\n", createdTime);

            char* modifiedTime = ctime(&node->file.inode.modified);
            modifiedTime[strlen(modifiedTime) - 1] = '\0'; // 줄바꿈 제거
            printf("수정 시간: %s\n", modifiedTime);

            printf("파일의 부모 디렉토리: %s\n", node->parent ? node->parent->dir.name : "없음");
            found = true;
        }
```

```
} else if (node->type == DIR_TYPE) {
    for (int i = 0; i < node->dir.childCount; i++) {
        Node* child = (Node*)node->dir.children[i];
        if (child->type == FILE_TYPE && strcmp(child->file.name, name) == 0) {
            printf("파일 이름: %s\n", child->file.name);
            printf("파일 내용: %s\n", child->file.content);
            printf("파일 크기: %d bytes\n", child->file.inode.fileSize);

            char* createdTime = ctime(&child->file.inode.created);
            createdTime[strlen(createdTime) - 1] = '\0'; // 줄바꿈 제거
            printf("생성 시간: %s\n", createdTime);

            char* modifiedTime = ctime(&child->file.inode.modified);
            modifiedTime[strlen(modifiedTime) - 1] = '\0'; // 줄바꿈 제거
            printf("수정 시간: %s\n", modifiedTime);

            printf("파일의 부모 디렉토리: %s\n", node->dir.name);
            found = true;
            break;
        } else if (child->type == DIR_TYPE) {
            readfile(child, name); // 재귀적으로 탐색
        }
    }
}
```

매개변수:

- **node**: 검색을 시작할 노드. 파일 또는 디렉토리
- **name**: 찾고자 하는 파일 이름

지정된 이름을 가진 파일을 찾고, 그 파일의 상세 정보를 출력한다. 디렉토리 내의 모든 파일 및 하위 디렉토리를 재귀적으로 탐색하여 파일의 부모 디렉토리, 내용, 크기, 생성/수정 시간을 출력한다. 파일이 존재하지 않을 경우 오류 메시지를 출력한다.

- **updatefile**

```
void updatefile(Node* parent, const char* name, const char* newContent) {
    if (parent->type != DIR_TYPE) {
        printf("'%s'는 디렉터리가 아닙니다.\n", parent->dir.name);
        return;
    }
    bool found = false;
    for (int i = 0; i < parent->dir.childCount; i++) {
        Node* child = (Node*)parent->dir.children[i];
        if (child->type == FILE_TYPE && strcmp(child->file.name, name) == 0) {
            // 파일 내용을 업데이트하고 수정 시간 갱신
            strncpy(child->file.content, newContent, sizeof(child->file.content) - 1);
            child->file.content[sizeof(child->file.content) - 1] = '\0'; // 안전하게 NULL 종료
            child->file.inode.fileSize = strlen(newContent);
            time(&child->file.inode.modified);

            printf("파일 '%s'의 내용이 업데이트 되었습니다.\n", name);
            printf("새로운 파일 내용: %s\n", child->file.content);
            printf("파일 크기: %d bytes\n", child->file.inode.fileSize);
```

```
            char* modifiedTime = ctime(&child->file.inode.modified);
            modifiedTime[strlen(modifiedTime) - 1] = '\0'; // 줄바꿈 제거
            printf("수정 시간: %s\n", modifiedTime);

            found = true;
            break;
        }
    }
    if (!found) {
        printf("'%s' 파일을 찾을 수 없습니다.\n", name);
    }
}
```

매개변수:

- **parent**: 파일이 속한 부모 디렉토리 노드
- **name**: 업데이트할 파일의 이름
- **newContent**: 파일에 새로 쓸 내용

지정된 이름을 가진 파일의 내용을 새로 주어진 내용으로 변경하고 파일의 수정 시간을 현재 시간으로 업데이트한다. 파일이 존재하지 않을 경우 오류 메시지를 출력한다.

- **searchfile**

```
void searchfile(Node* node, const char* keyword) {
    if (node->type == FILE_TYPE) {
        if (strstr(node->file.content, keyword) != NULL) {
            printf("키워드 '%s'를 포함하는 파일: %s\n", keyword, node->file.name);
            printf("해당 파일의 부모 디렉토리: %s\n", node->parent->dir.name);
            printf("파일 크기: %d바이트\n", node->file.inode.fileSize);

            // 생성 시간 출력
            char* createdTime = ctime(&node->file.inode.created);
            createdTime[strlen(createdTime) - 1] = '\0'; // 줄바꿈 제거
            printf("생성 시간: %s\n", createdTime);

            // 수정 시간 출력
            char* modifiedTime = ctime(&node->file.inode.modified);
            modifiedTime[strlen(modifiedTime) - 1] = '\0'; // 줄바꿈 제거
            printf("수정 시간: %s\n\n", modifiedTime);
        }
    } else if (node->type == DIR_TYPE) {
        for (int i = 0; i < node->dir.childCount; i++) {
            searchfile((Node*)node->dir.children[i], keyword);
        }
    }
}
```

매개변수:

- **node**: 검색을 시작할 노드. 파일 또는 디렉토리
- **keyword**: 파일 내용에서 찾고자 하는 키워드

지정된 키워드를 내용에 포함한 파일을 찾고, 그 파일의 이름, 부모 디렉터리 등의 상세 정보를 출력한다. 디렉토리 내의 모든 파일 및 하위 디렉토리를 재귀적으로 탐색한다.

- **hasChildWithName**

```
int hasChildWithName(Node* parent, const char* name, int type) {
    if (parent->type != DIR_TYPE) {
        return 0; // 부모가 디렉터리가 아니면 항상 0을 반환
    }
    for (int i = 0; i < parent->dir.childCount; i++) {
        Node* child = (Node*)parent->dir.children[i];
        if (type == DIR_TYPE && child->type == DIR_TYPE && strcmp(child->dir.name, name) == 0) {
            return 1; // 동일한 이름의 디렉터리 발견
        } else if (type == FILE_TYPE && child->type == FILE_TYPE && strcmp(child->file.name, name) == 0) {
            return 1; // 동일한 이름의 파일 발견
        }
    }
    return 0; // 동일한 이름의 자식 노드 없음
}
```

매개변수:

- **parent**: 부모 디렉토리 노드
- **name**: 찾고자 하는 자식 노드의 이름
- **type**: 찾고자 하는 자식 노드의 타입 (디렉토리 또는 파일)

부모 디렉토리의 자식 노드들을 탐색하여 사용자가 지정한 이름과 타입을 가진 자식 노드가 있는지 확인한다.

- **renameNode**

```
void renameNode(Node* parent, const char* oldName, const char* newName, NodeType type) {
    if (parent->type != DIR_TYPE) {
        printf("'%s'는 디렉터리가 아닙니다.\n", parent->dir.name);
        return;
    }
    // 같은 이름을 가진 자식 노드가 있는지 확인
    if (hasChildWithName(parent, newName, type)) {
        printf("'%s' 이름을 가진 %s가 이미 존재합니다.\n", newName, type == DIR_TYPE ? "디렉터리" : "파일");
        return;
    }

    bool found = false;
    for (int i = 0; i < parent->dir.childCount; i++) {
        Node* child = (Node*)parent->dir.children[i];
        if (child->type == type && strcmp(type == DIR_TYPE ? child->dir.name : child->file.name, oldName) == 0) {
            // 이름 변경
            strcpy(type == DIR_TYPE ? child->dir.name : child->file.name, newName);
            if (type == DIR_TYPE) {
                child->dir.inode.modified = time(NULL); // 노드 수정 시간 업데이트
            } else {
                child->file.inode.modified = time(NULL); // 노드 수정 시간 업데이트
            }
            printf("'%s'의 이름이 '%s'(으)로 변경되었습니다.\n", oldName, newName);
            found = true;
            break;
        }
    }
    if (!found) {
        printf("'%s'를 찾을 수 없습니다.\n", oldName);
    }
}
```

매개변수:

- **parent**: 부모 디렉토리 노드
- **oldName**: 변경할 파일 또는 디렉토리의 현재 이름
- **newName**: 새로 설정할 이름
- **type**: 노드의 타입 (디렉토리 또는 파일)

동일한 이름을 가진 자식 노드에 해당하는 파일 또는 디렉토리가 이미 존재하는지 확인한 뒤 지정된 이름을 가진 파일 혹은 디렉토리 이름을

변경하고 변경된 노드의 수정 시간을 현재 시간으로 업데이트한다. 새로운 이름의 노드가 이미 존재하는 경우, 부모 노드가 디렉터리가 아닌 경우, 노드를 찾지 못한 경우 에러 메시지를 출력한다.

- **deleteNode**

```
void deleteNode(Node* parent, const char* name, NodeType type) {
    if (parent->type != DIR_TYPE) {
        printf("'%s'는 디렉터리가 아닙니다.\n", parent->dir.name);
        return;
    }

    for (int i = 0; i < parent->dir.childCount; i++) {
        Node* child = (Node*)parent->dir.children[i];
        if (child->type == type && strcmp(type == DIR_TYPE ? child->dir.name : child->file.name, name) == 0) {
            // 자식 노드 삭제 처리
            freeTree(child);
            // 배열에서 삭제된 노드 제거
            for (int j = i; j < parent->dir.childCount - 1; j++) {
                parent->dir.children[j] = parent->dir.children[j + 1];
            }
            parent->dir.childCount--;
            printf("'%s' %s가 삭제되었습니다.\n", name, type == DIR_TYPE ? "디렉터리" : "파일");
            return;
        }
    }

    printf("'%s' %s를 찾을 수 없습니다.\n", name, type == DIR_TYPE ? "디렉터리" : "파일");
}
```

매개변수:

- **parent**: 자식 노드를 삭제할 부모 디렉토리 노드 포인터
- **name**: 삭제할 노드의 이름
- **type**: 삭제할 노드의 타입 (디렉토리 또는 파일)

freeTree 함수를 호출하여 주어진 이름과 타입을 가진 삭제할 노드와 그 자식 노드들의 메모리를 해제한다. 삭제된 노드를 부모 노드의 자식 배열에서 제거하고, 배열의 나머지 요소들을 앞으로 당긴다. 부모 노드가 디렉터리가 아닌 경우, 노드를 찾지 못한 경우 에러 메시지를 출력한다.

- **deepCopyNode**

```
void deepCopyNode(Node* original, Node* copy) {
    if (original->type == FILE_TYPE) {
        strcpy(copy->file.content, original->file.content);
        copy->file.inode.fileSize = original->file.inode.fileSize;
        copy->file.inode.created = time(NULL); // 복사 시점을 생성 시간으로 설정
        copy->file.inode.modified = time(NULL); // 복사 시점을 수정 시간으로 설정
        copy->file.inode.linkCount = 1; // 새 파일이므로 링크 수는 1
    } else if (original->type == DIR_TYPE) {
        for (int i = 0; i < original->dir.childCount; i++) {
            Node* child = (Node*)original->dir.children[i];
            Node* newChild = createNode(child->type == DIR_TYPE ? child->dir.name : child->file.name, child->type, copy);
            if (child->type == DIR_TYPE) {
                copy->dir.inode.fileSize = 0; // 자식 노드에 따라 달라질 수 있으므로, 0으로 초기화
                copy->dir.inode.created = time(NULL); // 복사 시점을 생성 시간으로 설정
                copy->dir.inode.modified = time(NULL); // 복사 시점을 수정 시간으로 설정
                copy->dir.inode.linkCount = original->dir.inode.linkCount; // 링크 수는 원본 디렉터리 링크 수와 동일하게 설정
                deepCopyNode(child, newChild);
            } else { // FILE_TYPE
                strcpy(newChild->file.content, child->file.content);
                copy->file.inode.fileSize = original->file.inode.fileSize;
                copy->file.inode.created = time(NULL); // 복사 시점을 생성 시간으로 설정
                copy->file.inode.modified = time(NULL); // 복사 시점을 수정 시간으로 설정
                copy->file.inode.linkCount = 1; // 새 파일이므로 링크 수는 1
            }
            addChild(copy, newChild);
        }
    }
}
```

매개변수:

- **original**: 복사할 원본 노드 포인터
- **copy**: 깊은 복사본을 저장할 노드 포인터

주어진 원본 노드의 깊은 복사본을 생성한다. 원본 노드의 타입이 파일인 경우, 파일의 내용과 메타데이터(크기, 생성 시간, 수정 시간, 링크 수)를 복사한다.

디렉토리 노드인 경우 자식 노드에 대해서도 깊은 복사를 수행하기 위해 재귀적으로 deepCopyNode 함수를 호출한다. time(NULL) 을 사용하여 복사 시점의 시간을 노드의 생성 및 수정 시간으로 설정하고 strcpy 함수를 사용하여 파일 내용을 복사한다.

- **copyNode**

```
void copyNode(Node* parent, const char* name, const char* newName, NodeType targetType, Node* targetParent) {
    if (parent->type != DIR_TYPE) {
        printf("'%s'는 디렉터리가 아닙니다.\n", parent->dir.name);
        return;
    }
    if (targetParent->type != DIR_TYPE) {
        printf("'%s'는 디렉터리가 아닙니다.\n", targetParent->dir.name);
        return;
    }
    if (hasChildWithName(targetParent, newName, targetType)) {
        printf("'%s' 이름을 가진 노드가 이미 '%s' 디렉터리에 존재합니다.\n", newName, targetParent->dir.name);
        return;
    }

    for (int i = 0; i < parent->dir.childCount; i++) {
        Node* child = (Node*)parent->dir.children[i];
        if ((child->type == targetType) && ((targetType == DIR_TYPE && strcmp(child->dir.name, name) == 0) || (targetType == FILE_TYPE && strcmp(child->file.name, name) == 0))) {
            Node* newCopy = createNode(newName, child->type, targetParent);
            deepCopyNode(child, newCopy);
            addChild(targetParent, newCopy);
            printf("'%s'가 '%s'(으)로 복사되었습니다.\n", name, newName);
            return;
        }
    }
    printf("'%s'를 찾을 수 없습니다.\n", name);
}
```

매개변수:

- **parent**: 복사할 노드가 있는 부모 디렉토리 노드 포인터
- **name**: 복사할 노드의 이름
- **newName**: 새 노드의 이름
- **targetType**: 복사할 노드의 타입
- **targetParent**: 복사본을 추가할 부모 디렉토리 노드 포인터

지정된 이름과 타입에 맞는 노드를 부모 디렉토리에서 찾고 deepCopyNode 함수를 사용하여 깊은 복사를 수행한다. addChild 함수를 사용하여 타겟이 된 부모 노드에 새 이름으로 복사본을 추가한다. 부모 노드와 타겟 부모 노드가 디렉터리가 아닌 경우, 타겟 부모 노드에 이미 동일한 이름의 노드가 존재하는 경우, 노드를 찾지 못한 경우 에러 메시지를 출력한다.

- **calculateDirectorySize**

```
void calculateDirectorySize(Node* node, int* totalSize) {
    if (node->type == FILE_TYPE) {
        *totalSize += node->file.inode.fileSize;
    } else if (node->type == DIR_TYPE) {
        int inodeIndex = node->dir.inodeIndex;
        *totalSize += inodeTable.inodes[inodeIndex].fileSize;
        for (int i = 0; i < node->dir.childCount; i++) {
            calculateDirectorySize((Node*)node->dir.children[i], totalSize);
        }
    }
}
```

매개변수:

- **node**: 크기를 계산할 디렉토리 노드 포인터
- **totalSize**: 총 크기를 저장할 정수 포인터

노드의 타입이 파일이면 파일 크기만 해당하고, 디렉토리인 경우 디렉토리 내 모든 자식 노드들의 크기를 합산하기 위해 재귀적으로 함수를 호출해 디렉토리의 총 크기를 계산한다.

- **printDirectorySize**

```
void printDirectorySize(Node* node) {
    if (node == NULL || node->type != DIR_TYPE) {
        printf("유효하지 않은 디렉터리 노드입니다.\n");
        return;
    }
    int totalSize = 0;
    calculateDirectorySize(node, &totalSize);
    printf("디렉터리 '%s'의 총 크기: %d bytes\n", node->dir.name, totalSize);
}
```

매개변수:

- **node**: 크기를 출력할 디렉토리 노드 포인터

calculateDirectorySize 함수를 사용하여 디렉토리의 총 크기를 계산하고 printf 함수를 사용하여 계산된 크기를 출력한다. 노드가 아니거나 타입이 디렉토리가 아닌 경우 에러 메시지를 출력하고 함수를 종료한다.

main 함수의 기능을 하는 void dir_main 함수는 파일 시스템의 기본적인 명령어들을 처리할 수 있는 반복문과 함께 디렉터리와 파일 생성, 읽기, 수정, 검색, 출력, 이름 변경, 삭제, 복사, 디렉터리 크기 출력 등의 기능을 제공한다. 코드의 각 부분에 대한 자세한 설명은 다음과 같다.

```
void dir_main() {
    Node* root = createNode("root", DIR_TYPE, NULL);

    superblock.totalInodes = MAX_INODES;
    superblock.usedInodes = 1;
    superblock.totalBlocks = 1000; // 임의의 값
    superblock.usedBlocks = 0;
    superblock.fileSystemSize = 1000000; // 임의의 값 (1MB)

    // InodeTable 초기화
    for (int i = 0; i < MAX_INODES; i++) {
        inodeTable.isAllocated[i] = false;
    }
    inodeTable.isAllocated[root->dir.inodeIndex] = true; // 루트 디렉터리 inode 사용 표시

    char command[100], name[100], parentName[100], content[1024];
```

Node라는 구조체는 파일 또는 디렉터리를 나타낸다. createNode 함수를 통해 파일 시스템의 최상위 디렉터리 역할을 하는 root 디렉터리 노드를 생성해 초기화하고 superblock 구조체에 파일 시스템 관련 정보를 초기화한다. inodeTable의 모든 inode를 사용 가능 상태로 표시하고, 루트 디렉터리 inode만 사용 중으로 표시한다.

```
    while (1) {
        printf("명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete, rename, copy, dirsize, quit): ");
        scanf("%s", command);

        if (strcmp(command, "quit") == 0) {
            break;
        } else if (strcmp(command, "makedir") == 0 || strcmp(command, "makefile") == 0) {
            printf("부모 디렉터리 이름: ");
            scanf("%s", parentName);
            Node* parentNode = findNode(root, parentName, DIR_TYPE);

            if (parentNode == NULL || parentNode->type != DIR_TYPE) {
                printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
                continue;
            }

            printf("이름: ");
            scanf("%s", name);

            // 여기에 추가된 부분: 같은 이름의 자식 노드가 있는지 검사
            if (strcmp(command, "makedir") == 0) {
                int foundDirectory = hasChildWithName(parentNode, name, DIR_TYPE);
                if (foundDirectory) {
                    printf("같은 이름의 디렉터리가 이미 존재합니다: %s\n", name);
                    continue; // 같은 이름의 노드가 있으면 명령을 건너뛴다
                }
            } else if (strcmp(command, "makefile") == 0) {
                int foundFile = hasChildWithName(parentNode, name, FILE_TYPE);
                if (foundFile) {
                    printf("같은 이름의 파일이 이미 존재합니다: %s\n", name);
                    continue; // 같은 이름의 노드가 있으면 명령을 건너뛴다
                }
            }
```

```
    if (strcmp(command, "makedir") == 0) {
        Node* newDir = createNode(name, DIR_TYPE, parentNode);
        addChild(parentNode, newDir);
        printf("디렉터리 '%s' 가 생성되었습니다.\n", name);
    } else { // makefile
        printf("파일 내용: ");
        scanf(" %[^\n]s", content); // 공백을 포함한 내용을 받기 위해 수정
        Node* newFile = createNode(name, FILE_TYPE, parentNode);
        strcpy(newFile->file.content, content);
        newFile->file.inode.fileSize = strlen(content); // 파일 크기 설정
        addChild(parentNode, newFile);
        updateFileContent(newFile, content);
        printf("파일 '%s' 가 생성되었습니다.\n", name);
    }
```

사용자로부터 명령어를 입력 받아 해당 명령어에 맞는 작업을 수행한다. strcmp 함수를 사용하여 입력 받은 문자열과 각 명령어 문자열을 비교한다. quit 명령어를 입력받으면 반복문을 벗어나 프로그램이 종료된다. makedir과 makefile 명령어를 사용하여 새로운 디렉터리 또는 파일을 생성한다. 부모 디렉터리를 찾고 해당 디렉터리 내에 중복 이름이 없는지 확인한 후, 새로운 노드를 생성하고 부모 노드에 연결한다.

```
    } else if (strcmp(command, "readfile") == 0) {
        printf("부모 디렉터리 이름: ");
        scanf("%s", parentName);
        Node* parentNode = findNode(root, parentName, DIR_TYPE);
        if (parentNode == NULL || parentNode->type != DIR_TYPE) {
            printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
            continue;
        }
        printf("파일 이름: ");
        scanf("%s", name);
        readfile(parentNode, name);
    } else if (strcmp(command, "updatefile") == 0) {
        printf("부모 디렉터리 이름: ");
        scanf("%s", parentName);
        Node* parentNode = findNode(root, parentName, DIR_TYPE);
        if (parentNode == NULL || parentNode->type != DIR_TYPE) {
            printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
            continue;
        }
        printf("파일 이름: ");
        scanf("%s", name);
        printf("새로운 파일 내용: ");
        scanf(" %[^\n]s", content); // 공백을 포함한 내용을 받기 위해 수정
        updatefile(parentNode, name, content);
    } else if (strcmp(command, "searchfile") == 0) {
        printf("검색할 키워드: ");
        scanf(" %[^\n]s", content); // 공백을 포함한 키워드를 받기 위해 수정
        searchfile(root, content);
```

readfile, updatefile, searchfile 명령어를 통해 파일의 내용을 읽거나 수정하고 키워드로 파일을 검색한다.

```
} else if (strcmp(command, "print") == 0) {
    printTree(root, 0);
} else if (strcmp(command, "rename") ==0) {
    char oldName[100];
    char newName[100];
    NodeType type;
    char typeName[10];

    printf("부모 디렉터리 이름: ");
    scanf("%s", parentName);
    Node* parentNode = findNode(root, parentName, DIR_TYPE);
    if (parentNode == NULL || parentNode->type != DIR_TYPE) {
        printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
        continue;
    }

    printf("변경할 파일/디렉터리의 이름: ");
    scanf("%s", oldName);
    printf("새 이름: ");
    scanf("%s", newName);
    printf("타입 ('file' 또는 'dir'): ");
    scanf("%s", typeName);
    type = strcmp(typeName, "dir") == 0 ? DIR_TYPE : FILE_TYPE;

    renameNode(parentNode, oldName, newName, type);
```

print, rename 명령어를 통해 현재 파일 시스템의 디렉터리 및 파일 구조를 트리 형태로 출력하거나 파일이나 디렉터리의 이름을 변경한다.

```
} else if (strcmp(command, "delete") == 0) {
    NodeType type;
    char typeName[10];

    printf("부모 디렉터리 이름: ");
    scanf("%s", parentName);
    Node* parentNode = findNode(root, parentName, DIR_TYPE);
    if (parentNode == NULL || parentNode->type != DIR_TYPE) {
        printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
        continue;
    }

    printf("삭제할 파일/디렉터리의 이름: ");
    scanf("%s", name);
    printf("타입 ('file' 또는 'dir'): ");
    scanf("%s", typeName);
    type = strcmp(typeName, "dir") == 0 ? DIR_TYPE : FILE_TYPE;

    deleteNode(parentNode, name, type);
```

```
} else if (strcmp(command, "copy") == 0) {
    char newName[100];
    char nodeName[100];
    NodeType type;
    char typeName[10];

    printf("부모 디렉터리 이름: ");
    scanf("%s", parentName);
    Node* parentNode = findNode(root, parentName, DIR_TYPE);
    if (parentNode == NULL || parentNode->type != DIR_TYPE) {
        printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
        continue;
    }

    printf("복사할 파일/디렉터리의 이름: ");
    scanf("%s", name);
    printf("타입 ('file' 또는 'dir'): ");
    scanf("%s", typeName);
    printf("어디에 복사할지. 디렉터리의 이름: ");
    scanf("%s", nodeName);
    printf("복사할 파일/디렉터리의 새 이름: ");
    scanf("%s", newName);
    type = strcmp(typeName, "dir") == 0 ? DIR_TYPE : FILE_TYPE;

    Node* newNode = findNode(root, nodeName, DIR_TYPE);

    copyNode(parentNode, name, newName, type, newNode);
```

delete, copy 명령어를 사용하여 파일이나 디렉터리의 이름을 변경하거나, 삭제, 또는 다른 위치로 복사한다.

```
    } else if(strcmp(command, "dirsize") == 0) {
        printf("부모 디렉터리 이름: ");
        scanf("%s", parentName);
        Node* parentNode = findNode(root, parentName, DIR_TYPE);
        if (parentNode == NULL || parentNode->type != DIR_TYPE) {
            printf("'%s' 디렉터리를 찾을 수 없습니다.\n", parentName);
            continue;
        }
        printDirectorySize(parentNode);
    }
    else {
        printf("알 수 없는 명령입니다.\n");
    }
}
```

dirsize 명령어를 통해 특정 디렉터리의 크기를 출력한다. 이는 해당 디렉터리에 포함된 모든 파일 및 하위 디렉터리의 크기를 합한 값이다.

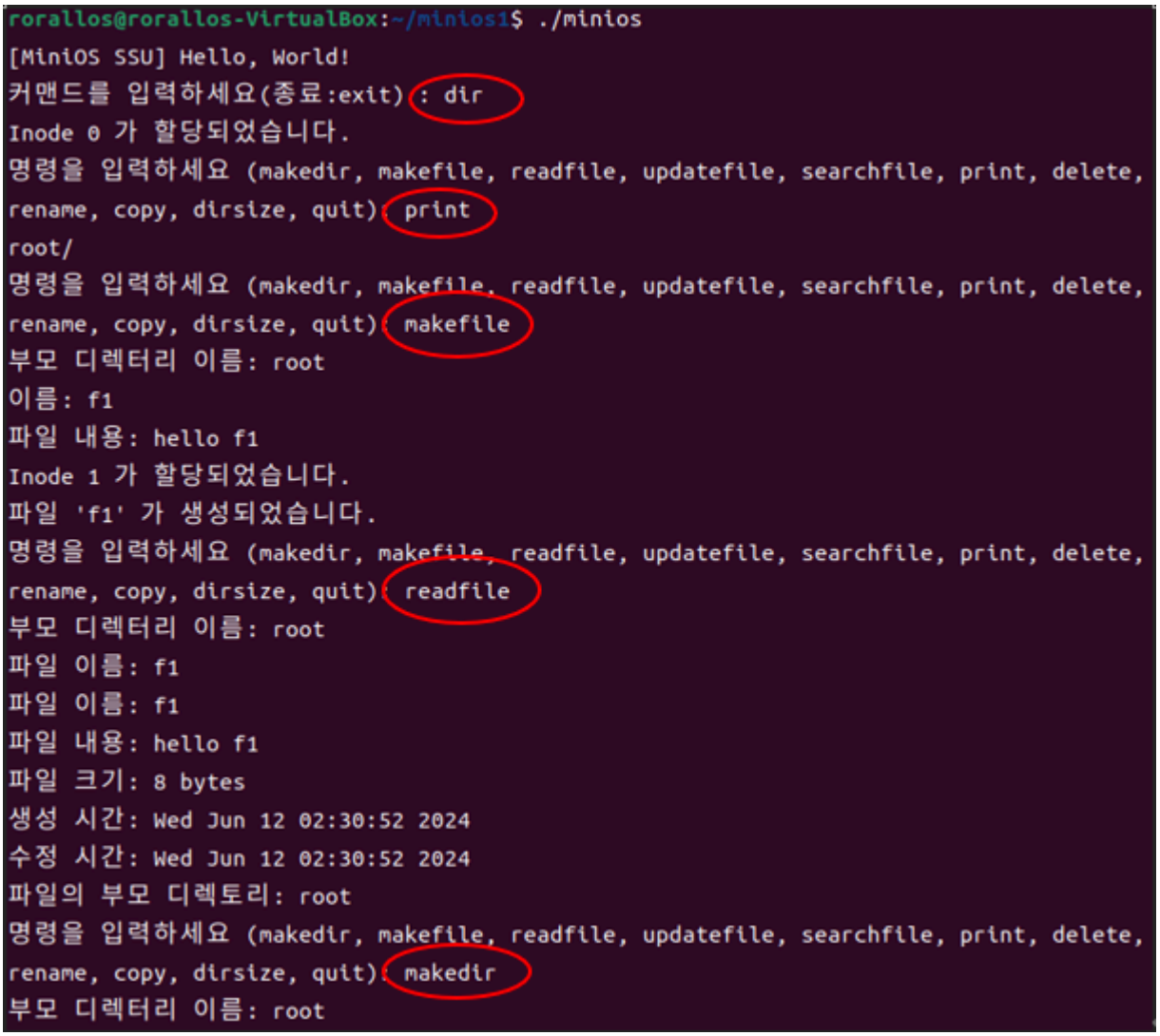
```
    printTree(root, 0);
    freeTree(root);

}
```

프로그램 종료 전 freeTree 함수를 사용하여 생성된 모든 노드의 메모리를 해제한다.

이 모듈은 기본적인 파일 시스템 구조를 시뮬레이션하며 디렉토리와 파일의 생성, 검색, 업데이트, 출력 및 해제를 포함한 다양한 기능을 제공한다. 각 함수는 트리 구조를 탐색하거나 수정하는 데 사용되며, 트리 구조를 활용하여 복잡한 데이터를 관리하는 방법을 실습하는 데 유용하다.

4. 테스트 및 검증


rorallos@rorallos-VirtualBox:~/minios1$ ./minios
[MiniOS SSU] Hello, World!
커맨드를 입력하세요(종료:exit) : dir
Inode 0 가 할당되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete, rename, copy, dirsize, quit) : print
root/
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete, rename, copy, dirsize, quit) : makefile
부모 디렉터리 이름: root
이름: f1
파일 내용: hello f1
Inode 1 가 할당되었습니다.
파일 'f1' 가 생성되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete, rename, copy, dirsize, quit) : readfile
부모 디렉터리 이름: root
파일 이름: f1
파일 이름: f1
파일 내용: hello f1
파일 크기: 8 bytes
생성 시간: Wed Jun 12 02:30:52 2024
수정 시간: Wed Jun 12 02:30:52 2024
파일의 부모 디렉토리: root
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete, rename, copy, dirsize, quit) : makedir
부모 디렉터리 이름: root

```
이름: dir1
Inode 2 가 할당되었습니다.
디렉터리 'dir1' 가 생성되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): makefile
부모 디렉터리 이름: dir1
이름: df1
파일 내용: hello...
Inode 3 가 할당되었습니다.
파일 'df1' 가 생성되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): dirsize
부모 디렉터리 이름: root
디렉터리 'root'의 총 크기: 16 bytes
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): makedir
부모 디렉터리 이름: root
이름: dir2
Inode 4 가 할당되었습니다.
디렉터리 'dir2' 가 생성되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): copy
부모 디렉터리 이름: root
복사할 파일/디렉터리의 이름: f1
타입 ('file' 또는 'dir'): file
어디에 복사할지. 디렉터리의 이름: dir2
복사할 파일/디렉터리의 새 이름: cf1
```

```
Inode 5 가 할당되었습니다.
'f1'가 'cf1'(으)로 복사되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): readfile
부모 디렉터리 이름: dir2
파일 이름: cf1
파일 이름: cf1
파일 내용: hello f1
파일 크기: 8 bytes
생성 시간: Wed Jun 12 02:35:28 2024
수정 시간: Wed Jun 12 02:35:28 2024
파일의 부모 디렉토리: dir2
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): updatefile
부모 디렉터리 이름: root
파일 이름: f1
새로운 파일 내용: he..i want to sleep..
파일 'f1'의 내용이 업데이트 되었습니다.
새로운 파일 내용: he..i want to sleep..
파일 크기: 21 bytes
수정 시간: Wed Jun 12 02:36:37 2024
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): searchfile
검색할 키워드: he
키워드 'he'를 포함하는 파일: f1
해당 파일의 부모 디렉토리: root
파일 크기: 21바이트
```

```
생성 시간: Wed Jun 12 02:30:52 2024
수정 시간: Wed Jun 12 02:36:37 2024

키워드 'he'를 포함하는 파일: df1
해당 파일의 부모 디렉토리: dir1
파일 크기: 8바이트
생성 시간: Wed Jun 12 02:33:57 2024
수정 시간: Wed Jun 12 02:33:57 2024

키워드 'he'를 포함하는 파일: cf1
해당 파일의 부모 디렉토리: dir2
파일 크기: 8바이트
생성 시간: Wed Jun 12 02:35:28 2024
수정 시간: Wed Jun 12 02:35:28 2024

명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): print
root/
  f1
  dir1/
    df1
  dir2/
    cf1
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): rename
부모 디렉터리 이름: dir2
변경할 파일/디렉터리의 이름: cf1
```

```
새 이름: rcf1
타입 ('file' 또는 'dir'): file
'cf1'의 이름이 'rcf1'(으)로 변경되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): copy
부모 디렉터리 이름: dir2
복사할 파일/디렉터리의 이름: rcf1
타입 ('file' 또는 'dir'): file
어디에 복사할지. 디렉터리의 이름: dir2
복사할 파일/디렉터리의 새 이름: ccf1
Inode 6 가 할당되었습니다.
'rcf1'가 'ccf1'(으)로 복사되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): readfile
부모 디렉터리 이름: dir2
파일 이름: ccf1
파일 이름: ccf1
파일 내용: hello f1
파일 크기: 8 bytes
생성 시간: Wed Jun 12 02:38:16 2024
수정 시간: Wed Jun 12 02:38:16 2024
파일의 부모 디렉토리: dir2
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): delete
부모 디렉터리 이름: dir2
삭제할 파일/디렉터리의 이름: rcf1
타입 ('file' 또는 'dir'): dir
```

```
'rcf1' 디렉터리를 찾을 수 없습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): dirsize
부모 디렉터리 이름: dir1
디렉터리 'dir1'의 총 크기: 8 bytes
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): print
root/
  f1
  dir1/
    df1
  dir2/
    rcf1
    ccf1
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): delete
부모 디렉터리 이름: dir1
삭제할 파일/디렉터리의 이름: df1
타입 ('file' 또는 'dir'): file
Inode 3 가 해제되었습니다.
'df1' 파일가 삭제되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): dirsize
부모 디렉터리 이름: root
디렉터리 'root'의 총 크기: 37 bytes
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): dirsize
```

```
부모 디렉터리 이름: dir1
디렉터리 'dir1'의 총 크기: 0 bytes
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): rename
부모 디렉터리 이름: dir2
변경할 파일/디렉터리의 이름: ccf1
새 이름: rcf1
타입 ('file' 또는 'dir'): file
'rcf1' 이름을 가진 파일가 이미 존재합니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): rename
부모 디렉터리 이름: dir2
변경할 파일/디렉터리의 이름: ccf1
새 이름: cccf
타입 ('file' 또는 'dir'): file
'ccf1'의 이름이 'cccf'(으)로 변경되었습니다.
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): print
root/
  f1
  dir1/
  dir2/
    rcf1
    cccf
명령을 입력하세요 (makedir, makefile, readfile, updatefile, searchfile, print, delete,
rename, copy, dirsize, quit): quit
root/
```

```
  f1
  dir1/
  dir2/
    rcf1
    cccf
Inode 1 가 해제되었습니다.
Inode 2 가 해제되었습니다.
Inode 5 가 해제되었습니다.
Inode 6 가 해제되었습니다.
Inode 4 가 해제되었습니다.
Inode 0 가 해제되었습니다.
커맨드를 입력하세요(종료:exit) :
```

5. 완료 사항

inode 기반 파일 시스템을 miniOS에 구현 완료한 사항은 다음과 같다.

1. 디스크 구조 설계
- 디스크는 슈퍼 블록, inode 테이블로 구성.
- 슈퍼 블록에는 파일 시스템의 메타데이터(총 블록 수, 사용 가능한 블록 수, inode 수 등)가 저장됨.
- inode 테이블에는 각 파일과 디렉토리에 대한 inode가 저장됨.

2. inode 구조 설계
- 각 inode는 파일의 메타데이터를 포함(파일 크기, 생성/수정 시간 등).

3. 파일 및 디렉토리 관리
  - 파일 생성, 삭제, 읽기, 쓰기 기능 구현.
  - 디렉토리 생성, 삭제, 탐색 기능 구현.
  - 경로를 통해 파일과 디렉토리를 접근할 수 있도록 간접적인 경로 파싱 기능 구현.

4. 블록 할당 및 관리
  - inode 할당을 위한 프리 리스트(free list) 관리.

이와 같은 단계들을 통해 miniOS에서 inode 기반 파일 시스템을 구현하고 완료하였다.

6. 예정사항

데이터의 안정성과 신뢰성을 보장하고자 파일 시스템의 무결성을 유지하고 오류를 감지 및 복구할 수 있는 메커니즘을 구현할 것이다. 다음과 같은 모듈을 도입한다.

Consistency Checker

Consistency Checker는 파일 시스템의 무결성을 주기적으로 검사하고, 오류가 발견되면 이를 복구하는 역할을 한다. 이 모듈은 다음과 같은 기능을 포함한다.

1. 무결성 검사
- 파일 시스템의 메타데이터를 주기적으로 검사하여 일관성을 확인한다.
- 각 inode의 상태를 점검하고, 파일 시스템 구조의 일관성을 유지한다.

2. 오류 감지
- 손상된 inode 식별한다.
- 파일 시스템의 구조적 불일치, 메타데이터의 오류 등을 감지한다.

3. 오류 복구
- 감지된 오류를 기반으로 가능한 경우 자동으로 복구를 시도한다.
- 손상된 inode를 재구성하여 파일 시스템의 정상적인 운영을 유지한다.
- 복구가 불가능한 경우, 손상된 부분을 격리하고 사용자에게 알린다.

Consistency Checker는 파일 시스템의 무결성을 유지하고, 데이터 손실을 최소화하며, 시스템의 안정성을 보장하는 중요한 역할을 할 것이다. 주기적인 검사를 통해 잠재적인 문제를 조기에 발견하고, 신속한 복구를 통해 사용자 데이터를 보호한다.

이러한 메커니즘은 파일 시스템의 신뢰성을 높이고, 예기치 않은 오류로 인한 데이터 손실 위험을 줄이는 데 기여할 것이다. Consistency Checker를 통해 파일 시스템의 무결성을 지속적으로 유지함으로써, 안정적이고 신뢰할 수 있는 데이터 저장 환경을 제공한다.

7. 문제해결

우리 조는 'inode를 이용한 파일 시스템 구현'이라는 주제에 도달하기까지 많은 어려움을 겪었다.
첫 주제는 '세마포어와 뮤텍스 구현 및 비교'를 통해 각 동기화 상황에서 세마포어와 뮤텍스의 작동방식과 성능을 비교하고자 하였다. 하지만 주제가 다른 조들과 비교했을 때 약하다는 중간 피드백을 받았다.
그래서 프로세스 스케줄링 후 프로세스가 메모리에 접근하여 시스템 콜을 통한 파일시스템을 구현하고자 했는데, 메모리 접근과 관련하여 여러 기술적 어려움이 발생하였고, 프로세스 스케줄링과 파일시스템 명령어의 결합하는 부분도 어려움을 겪었기에 결국 주제를 두 번 변경한 끝에 최종적으로 선택한 주제였기 때문에 구현에 필요한 시간도 일주일 정도로 매우 촉박했다. 그럼에도 불구하고 팀원 모두가 협동하여 문제를 해결하고 최종적으로 파일 시스템을 구현할 수 있었다.

기술적으로는 다음과 같은 어려움이 있었다.

1. Segmentation Fault 문제

파일 시스템을 구현하는 과정에서 메모리 접근과 관련하여 지속적으로 segmentation fault가 발생했다. 이 문제를 해결하는 데 2주 동안의 시간을 소비했지만, 해결되지 않아 결국 주제를 변경해야 했다.

2. 디렉토리 사이즈 측정 함수와 업데이트 함수 충돌

아이노드 테이블과 슈퍼블록을 추가하는 과정에서 디렉토리 사이즈를 측정하는 함수와 이를 업데이트하는 함수를 추가했는데, 이 두 함수가 같은 구조체 변수를 업데이트할 때 동기화가 잘 이루어지지 않는 문제가 발생했다. 결국 업데이트 함수를 제거하고 사이즈 측정 함수만 사용하게 되었다. 이로 인해 실시간 업데이트 기능이 없어져 아쉬운 부분이 있었다.

3. inode 초기화 문제

dir_main 함수에서 아이노드와 관련된 구조체 변수들을 초기화하는 과정에서 실수가 발생하여 종료 시 inode0이 두 번 해제되는 문제가 있었다. 이 문제는 변수의 초기화 설정을 바꿔줌으로써 해결할 수 있었다.

4. 이름이 중복될 때

트리구조의 특성상 함수를 실행할 때 왼쪽에서 부터 순서대로 노드를 검색한다. 이때 중복된 이름이 있다면 내가 원하는 파일이 아닌 다른 파일을 수정하거나 복사하는 경우가 생긴다. 따라서 특정 함수를 실행하기 전 어떤 부모 디렉토리 밑에서 함수를 실행할 것인지 명시를 해줌으로써 해결했다.

5. 프로세스 스케줄링 무한 대기 상태

프로세스를 생산해 프로세스들이 메모리에서 명령어에 접근해 파일 시스템의 명령어를 수행하는 식으로 모듈을 구성하려 했으나, 아무 것도 없는 miniOS 상황에서 무수히 많은 프로세스를 생산해내기가 힘들었고 프로세스가 비어 대기상태에 놓이지만 이 상태에서 할 수 있는게 없어 결국 무한대기상태에 빠지게 되었다. 거기다 시스템 콜은 명령어는 원래 메모리에서 읽어오는 식이어야 하지만 해당 파일시스템은 실제로

메모리에서 읽어오는 식도 아닌지라 원래 있던 프로세스 스케줄링 방식이랑 상충되는 부분도 있었다.

위와 같은 어려운 점들을 겪었지만, 팀원 모두가 협력하고 문제를 해결해 나가면서 결국 파일 시스템을 성공적으로 구현할 수 있었다. 이러한 경험은 기술적 역량뿐만 아니라 팀워크와 문제 해결 능력을 크게 향상시킬 수 있는 소중한 기회였다.

[역할 분담 및 수행일지]

팀원	김석주(팀장)	김지현	김진한	지주원
역할	세마포어 함수 구현 및 동기화 테스트 시스템 콜 활용한 파일 시스템 구현 및 시뮬레이션 파일시스템에 접근하는 프로세스 스케줄링 구축 시도	뮤텍스 함수 구현 및 동기화 테스트 시스템 콜 활용한 파일 시스템 구현 및 시뮬레이션 inode를 이용한 파일 시스템 수정 보고서 작성	세마포어 함수 구현 및 동기화 테스트 시스템 콜 활용한 파일 시스템 구현 및 시뮬레이션 inode를 이용한 파일 시스템 구현	뮤텍스 함수 구현 및 동기화 테스트 시스템 콜 활용한 파일 시스템 구현 및 시뮬레이션 구축된 모든 miniOS 시뮬레이션 진행 보고서 작성
회의 일자	회의 내용			
4/17	프로젝트 주제 토의			
4/24	프로젝트 주제 토의 및 세마포어, 뮤텍스 함수 구현			
5/1	동기화 테스트 시나리오 선정 및 구현			
5/8	세마포어와 뮤텍스 성능 비교, 분석			
5/15	시스템 콜, 파일 시스템 공부			
5/22	중간 보고서 작성, 시스템 콜을 통한 파일 시스템 구현			
5/29	메모리 접근 에러 해결 x			
6/5	inode를 이용한 파일 시스템 구현			

6/12	최종 보고서 작성 및 최종 발표 준비
6/13	발표 준비